**Database Basics**

**✅ SECTION A: What Is a Database?**

A **Database** is an organized collection of data that can be easily accessed, managed, and updated.

**🧠 Types of Databases:**

1. **Relational** (SQL):
   * Examples: PostgreSQL, MySQL, SQLite
   * Structured data in tables with rows and columns
2. **Non-Relational** (NoSQL):
   * Examples: MongoDB, Redis, Cassandra
   * Flexible schema; JSON-like docs, key-value stores, etc.

**✅ SECTION B: Why Use Databases in Python?**

Python is often used to:

* ✅ Store form/user input
* ✅ Query logs or metrics
* ✅ Build web apps (Django, Flask, FastAPI)
* ✅ Interact with CI/CD audit logs or pipeline data

**✅ SECTION C: Common SQL Concepts You Must Know**

| **Term** | **Meaning** |
| --- | --- |
| Table | Collection of structured rows (like Excel sheet) |
| Row | One record (a single entry) |
| Column | One data field (e.g., username, email) |
| Primary Key | Unique identifier for a row |
| Query | Command to interact with the database (SQL command) |

**✅ SECTION D: Let’s Start with SQLite (No Install Needed)**

**Why SQLite?**

* ✔️ Built-in in Python (via sqlite3 module)
* ✔️ Lightweight for demos, prototypes
* ✔️ File-based — no server needed

**✅ Code Example 1: Create and Connect to a SQLite Database**

import sqlite3

# Connect to a database file (creates if not exists)

conn = sqlite3.connect("test.db")

# Create a cursor to execute SQL commands

cursor = conn.cursor()

**🔍 Explanation:**

* sqlite3.connect("test.db")  
  → Creates or opens a database file called test.db
* conn.cursor()  
  → A cursor is like a command executor in SQL — we run queries using it.

✅ No server or credentials needed.

**✅ SECTION E: Creating a Table**

**✅ Code Example 2: Creating a Table in SQL**

cursor.execute('''

CREATE TABLE IF NOT EXISTS users (

id INTEGER PRIMARY KEY AUTOINCREMENT,

name TEXT NOT NULL,

email TEXT UNIQUE NOT NULL

)

''')

**🔍 Explanation:**

* CREATE TABLE is SQL syntax for creating a new table.
* id INTEGER PRIMARY KEY AUTOINCREMENT  
  → Unique row ID, automatically increases
* TEXT → data type for name/email
* NOT NULL → must be provided
* UNIQUE → email must be distinct

✅ Use IF NOT EXISTS to avoid errors if table already exists.

**✅ SECTION F: Inserting Data into the Table**

**✅ Code Example 3: Insert Records**

cursor.execute("INSERT INTO users (name, email) VALUES (?, ?)", ("Alice", "alice@example.com"))

cursor.execute("INSERT INTO users (name, email) VALUES (?, ?)", ("Bob", "bob@example.com"))

conn.commit()

**🔍 Explanation:**

* VALUES (?, ?)  
  → Placeholder for safe variable substitution (avoids SQL injection)
* ("Alice", "alice@example.com")  
  → Tuple of actual values to insert
* conn.commit()  
  → Saves changes to the DB file

✅ Always commit after writing to the DB.

**✅ SECTION G: Reading from the Database**

**✅ Code Example 4: Select and Print Data**

cursor.execute("SELECT \* FROM users")

rows = cursor.fetchall()

for row in rows:

print(row)

**🔍 Explanation:**

* SELECT \* FROM users  
  → Fetches all rows and all columns
* .fetchall() → returns list of row tuples
* Loop prints each record

✅ Output:

(1, 'Alice', 'alice@example.com')

(2, 'Bob', 'bob@example.com')

**✅ SECTION H: Updating and Deleting Records**

**✅ Code Example 5: Update a User**

cursor.execute("UPDATE users SET email = ? WHERE name = ?", ("alice@devopsshack.com", "Alice"))

conn.commit()

**✅ Code Example 6: Delete a User**

cursor.execute("DELETE FROM users WHERE name = ?", ("Bob",))

conn.commit()

**🔍 Explanation:**

* UPDATE users SET column = value WHERE condition
* DELETE FROM users WHERE condition
* Always use placeholders (?) for safe queries

✅ After both, call .commit() to persist.

**✅ SECTION I: Closing the Connection**

conn.close()

✅ Always close the connection when done.

**🧪 BONUS: Use SQLite in Memory (No File Created)**

conn = sqlite3.connect(":memory:")

🧠 Great for testing — no data saved on disk.

**🔚 PART 1 WRAP-UP: SQLite Quick Reference**

| **Operation** | **SQL Command** |
| --- | --- |
| Create table | CREATE TABLE |
| Insert data | INSERT INTO |
| Read data | SELECT \* FROM table |
| Update data | UPDATE table SET col = val |
| Delete data | DELETE FROM table WHERE condition |

**✅ SECTION A: Querying Specific Records Using WHERE**

Let’s say we have this table users:

| **id** | **name** | **email** |
| --- | --- | --- |
| 1 | Alice | alice@devopsshack.com |
| 2 | Bob | bob@example.com |
| 3 | Charlie | charlie@example.com |

**✅ Code Example 1: Query a User by Name**

import sqlite3

conn = sqlite3.connect("test.db")

cursor = conn.cursor()

cursor.execute("SELECT \* FROM users WHERE name = ?", ("Alice",))

user = cursor.fetchone()

print("User Found:", user)

**🔍 Explanation:**

* SELECT \* FROM users WHERE name = ?  
  → SQL command that filters rows by name
* ("Alice",)  
  → Tuple with one element (note the **comma**!)
* fetchone()  
  → Returns the **first match** (or None)

✅ Output:

User Found: (1, 'Alice', 'alice@devopsshack.com')

✅ Avoids SQL injection risk using ? + tuple

**✅ SECTION B: Use LIKE for Pattern Matching**

**✅ Code Example 2: Find All Users with Email from example.com**

cursor.execute("SELECT \* FROM users WHERE email LIKE ?", ("%@example.com",))

results = cursor.fetchall()

for user in results:

print("Match:", user)

**🔍 Explanation:**

* LIKE is used in SQL for pattern match.
* % is a wildcard:
  + %@example.com → ends with @example.com
* fetchall() returns a list of tuples

✅ Output:

Match: (2, 'Bob', 'bob@example.com')

Match: (3, 'Charlie', 'charlie@example.com')

**✅ SECTION C: Limiting the Number of Results with LIMIT**

**✅ Code Example 3: Get First 2 Records Only**

cursor.execute("SELECT \* FROM users LIMIT 2")

print(cursor.fetchall())

✅ Output:

[(1, 'Alice', 'alice@devopsshack.com'), (2, 'Bob', 'bob@example.com')]

💡 Useful for:

* Pagination
* Previewing data
* Avoiding memory issues on large datasets

**✅ SECTION D: Parameterized Queries from User Input (Simulated)**

**✅ Code Example 4: Simulate User Search**

user\_input = "Charlie"

cursor.execute("SELECT email FROM users WHERE name = ?", (user\_input,))

result = cursor.fetchone()

if result:

print("📧 Email:", result[0])

else:

print("❌ No such user")

**🔍 Explanation:**

* The input is safely injected into SQL query using placeholders
* Prevents risk from malicious input like ' OR 1=1 --

✅ Output:

📧 Email: charlie@example.com

**✅ SECTION E: Fetch Results as Python Dictionary (for DevOps Logging / APIs)**

Normally, fetchall() returns a list of **tuples**.  
But for more meaningful access, we can convert each row to a **dict**.

**✅ Code Example 5: Return Query Results as Dictionary**

conn.row\_factory = sqlite3.Row

cursor = conn.cursor()

cursor.execute("SELECT \* FROM users")

rows = cursor.fetchall()

for row in rows:

print(dict(row))

**🔍 Explanation:**

* row\_factory = sqlite3.Row  
  → Makes each row act like a dictionary
* dict(row) → Easily convert for JSON, logging, etc.

✅ Output:

{'id': 1, 'name': 'Alice', 'email': 'alice@devopsshack.com'}

{'id': 2, 'name': 'Bob', 'email': 'bob@example.com'}

{'id': 3, 'name': 'Charlie', 'email': 'charlie@example.com'}

**✅ SECTION F: Dynamic Query Construction (Advanced Tip)**

**✅ Code Example 6: Query Based on Dynamic Field**

def search\_user(field, value):

query = f"SELECT \* FROM users WHERE {field} = ?"

cursor.execute(query, (value,))

return cursor.fetchall()

result = search\_user("email", "bob@example.com")

print(result)

**🔍 Explanation:**

* Allows flexible querying on different columns
* Uses string formatting **only for column name**, not values

🚨 Security Tip: NEVER allow unsanitized user input into field. Use a whitelist check.